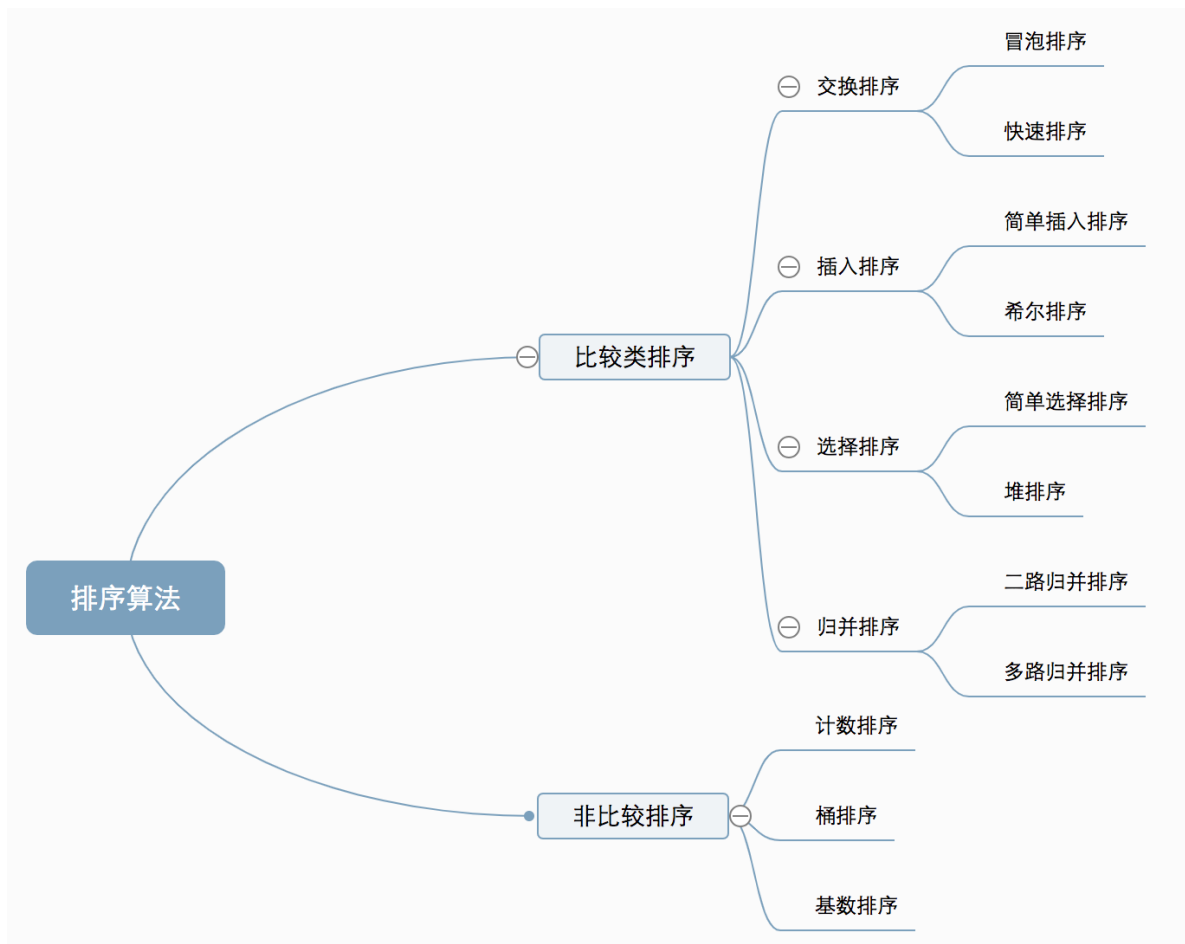


一、排序算法

1.0 算法分类



十种常见排序算法可以分为两大类：

- **比较类排序**：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破 $O(n\log n)$ ，因此也称为非线性时间比较类排序。
- **非比较类排序**：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排序。

1.1 各算法复杂度

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

1.2 选择排序

//1.从第一个元素开始，比较当前元素与剩下元素中最小的值的大小，如果比剩余元素最小的值大，那么交换两个元素的位置

//2.依次向后类推，直到最后一个元素

//3.选择排序是一个不稳定的排序，比如说数组为{5,5,2,8}，那么第一个5首先与2交换位置，这样原来的第一个5和第二个5的先后次序就颠倒了，所以是不稳定排序

选择排序工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。（简单说就是每次都从未排序区选择一个最大或最小的放入排序区）

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为R[1..n]，有序区为空；
- 第i趟排序(i=1,2,3...n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n)。该趟排序从当前无序区中-选出关键字最小

的记录 $R[k]$ ，将它与无序区的第1个记录 R 交换，使 $R[1..i]$ 和 $R[i+1..n]$ 分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；

- $n-1$ 趟结束，数组有序化了。

1.3 冒泡排序

冒泡排序(空复杂: $O(1)$ 时最好:序列中元素原本都是有序的, 所以 $O(n)$, 时最坏序列中元素原本都是逆序的: $O(n^2)$, 时平均: $O(n^2)$)

思路: 重复地遍历要排序的序列, 一次比较相邻的两个元素大小, 如果他们的大小顺序错误将他们交换。遍历数列的工作是重复进行的直到没有再需要交换的元素为止, 这个时候序列就已经排序完成了。

步骤:

//1.比较相邻的元素, 若第一个元素比第二个大, 就交换它们两个

//2.对每一对相邻的元素做同样的操作, 从前向后, 这样完成一圈, 最后的元素就是最大的数 (就把最大的数放到了最后面)

//3.每次对越来越少的元素重复以上步骤, 直到没有一个数字需要比较。

冒泡中两相邻元素若相等不会交换位置, 相同元素前后顺序在排序前后不改变, 所以稳定。

1.4 ★快速排序

快速排序(最好: $O(n \log n)$, 最坏(初始序列本身是有序或逆序): $O(n^2)$, 平均: $O(n \log n)$, 空间: $O(\log n)$)

快排是利用分治的思想, (1)首先从待排序序列中挑一个元素作为基准(2)接下来定义两个下标索引 low 和 $high$ 分别指向序列首部元素和尾部元素, low 从左向右遍历, 直到遇到比基准元素大的元素停止遍历, $high$ 从右向左遍历直到遇到比基准元素小的元素停止遍历, 然后交换 low 和 $high$ 所指的元素, 最后所有比基准值小的元素放在基

基准值前面，大于等于基准值的放在他后面，当low和high相遇，交换基准元素和low所指的元素，基准值就处于序列的中间位置，整个区间就划分成了两个独立的子区间。(3)然后分别递归的对两个独立的子区间重复上述过程，直到最后划分的子区间内只有一个元素为止。

快排不稳定

快速排序代码递归法(力扣40):

```
void quickSort(vector<int>& arr, int l, int r) {  
    // 子数组长度为 1 时终止递归  
    if (l >= r) return;  
    // 哨兵划分操作（以 arr[l] 作为基准数）  
    int i = l, j = r;  
    while (i < j) {  
        while (i < j && arr[j] >= arr[l]) j--;  
        while (i < j && arr[i] <= arr[l]) i++;  
        swap(arr[i], arr[j]);  
    }  
    swap(arr[i], arr[l]);  
    // 递归左（右）子数组执行哨兵划分  
    quickSort(arr, l, i - 1);  
    quickSort(arr, i + 1, r);  
}
```

arr: 2 4 1 0 3 5

^ ^
i j
l r

```

i, j = l, r
while i < j:
    while i < j and arr[j] >= arr[l]: j -= 1
    while i < j and arr[i] <= arr[l]: i += 1
    arr[i], arr[j] = arr[j], arr[i]
arr[l], arr[i] = arr[i], arr[l]

```

初始化“哨兵”索引位置，以 arr[l] 为基准数
循环交换，两哨兵相遇时跳出
从右向左 查找 首个小于基准数的元素
从左向右 查找 首个大于基准数的元素
交换 arr[i] 和 arr[j]
交换基准数 arr[l] 和 arr[i]

递归：对 **左子数组** 和 **右子数组** 递归执行 **哨兵划分**，直至子数组长度为 1 时终止递归，即可完成对整个数组的排序。

如下图所示，为示例数组 [2, 4, 1, 0, 3, 5] 的快速排序流程。观察发现，快速排序和 **二分法** 的原理类似，都是以 \log 时间复杂度实现搜索区间缩小。

1.5 归并排序

归并排序(最好、最坏、平均时复杂都是: $O(n\log n)$ ，空复杂: $O(n)$)
稳定排序

(1)使用两个数组索引分别指向待排序序列A的首尾元素，指明排序的范围，使用 $mid = (low + high) / 2$ ，将A拆成左右相邻的两部分，左半部分 $[low, mid]$ ，右半部分 $[mid + 1, high]$ 。

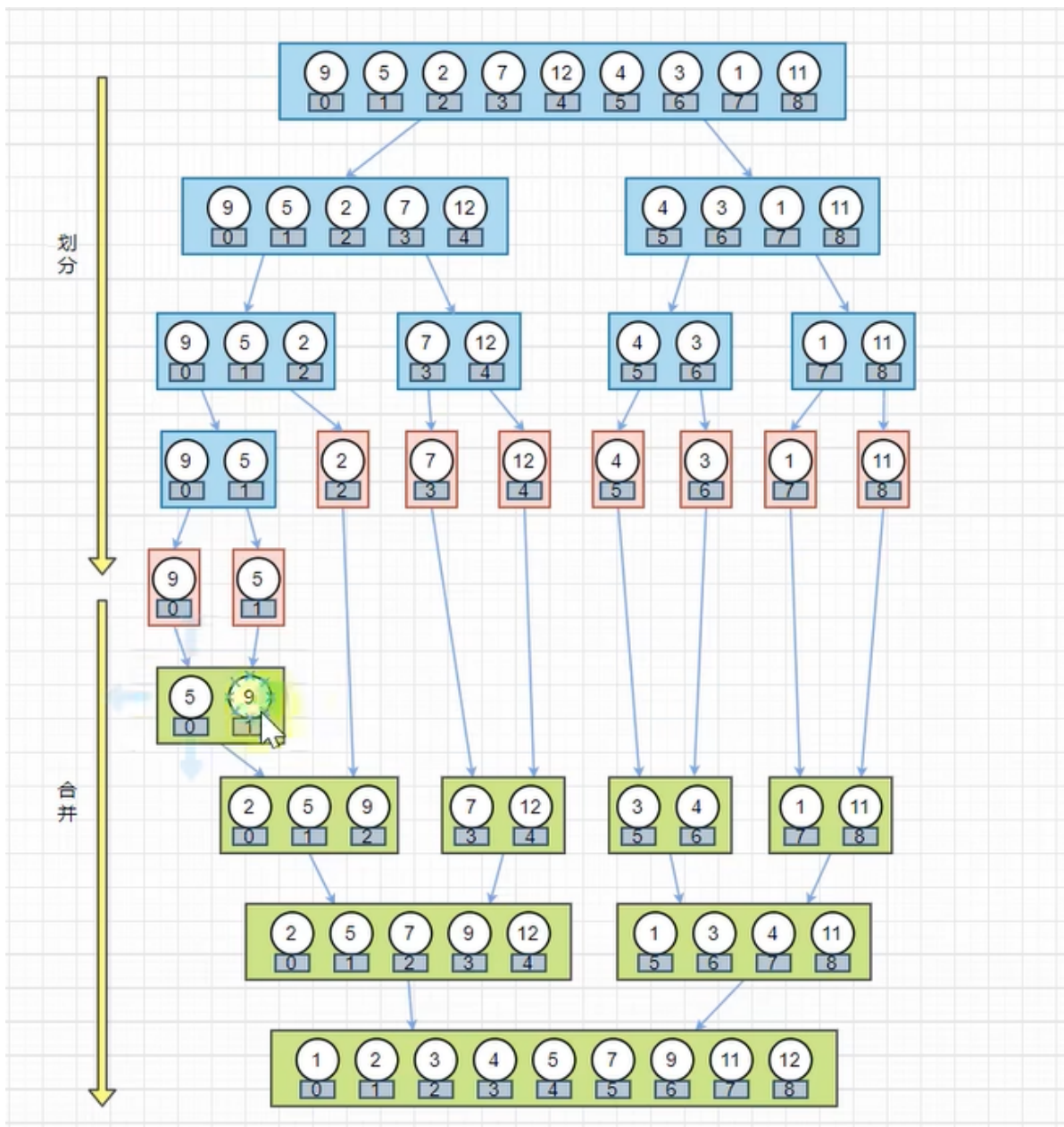
(2)对左半部分和右半部分递归的进行归并排序，经过递归处理后，左右两个子序列都变成了有序序列。(两个子区间分别递归到只剩一个元素，再归并成有序数组)

(3)接下来就是具体归并左右两个有序的子序列：定义辅助数组B，其大小与A相同，把A中元素全部复制到B中，用两个数组下标分别指向B中两个有序子序列的初始位置，比较两个数组下标所指元素大小，选择较小的元素复制回A中，将数组下标向后移动一位，重复上一步骤，直到某一个数组下标移动到子序列尾了，这时候直接将另一序列剩下的所有元素直接复制到A序列尾部。

合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性，所以归并排序稳定。

```
int *B=(int *)malloc(n*sizeof(int)); //辅助数组B
//A[low...mid]和A[mid+1...high]各自有序，将两个部分归并
void Merge(int A[],int low,int mid,int high){
    int i,j,k;
    for(k=low;k<=high;k++)
        B[k]=A[k]; //将A中所有元素复制到B中
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[i]<=B[j])
            A[k]=B[i++]; //将较小值复制到A中
        else
            A[k]=B[j++];
    }//for
    while(i<=mid) A[k++]=B[i++];
    while(j<=high) A[k++]=B[j++];
}

void MergeSort(int A[],int low,int high){
    if(low<high){
        int mid=(low+high)/2; //从中间划分
        MergeSort(A,low,mid); //对左半部分归并排序
        MergeSort(A,mid+1,high); //对右半部分归并排序
        Merge(A,low,mid,high); //归并
    }//if
}
```



1.6 堆排序

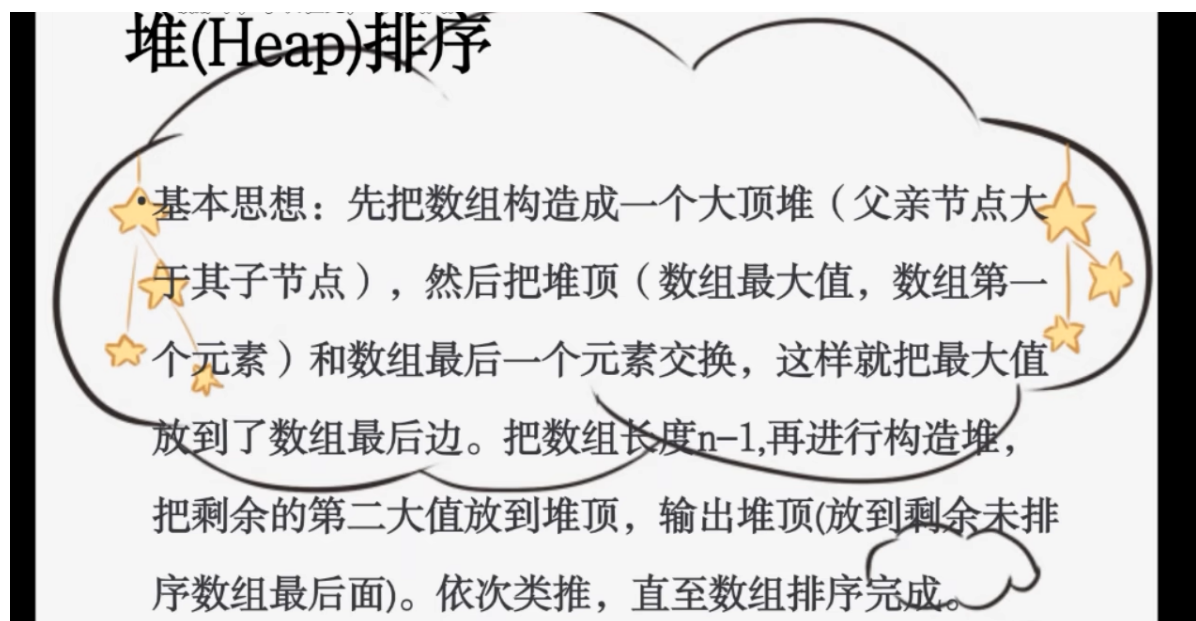
堆排序(时间 $O(n\log n)$ 、空间 $O(1)$ 、不稳定)

(1)将初始待排序序列($R_1 \dots R_n$)构建成大顶堆，此堆初始时各元素在数组中的存放是无序的

(2)将堆顶元素最大的元素 $R[1]$ 与最后一个元素 $R[n]$ 交换，此时可将数组中的堆元素划分为无序区(R_1, R_2, \dots, R_{n-1})和有序区(R_n),且满足 $R[1, 2 \dots n-1]$ 的值 $\leq R[n]$ 的值;

(3)由于交换后新的堆顶 $R[1]$ 可能违反大根堆的性质，因此需要对当前无序区 $(R1,R2,.....Rn-1)$ 调整为新堆，满足大根堆的性质后再次将 $R[1]$ 与无序区最后一个元素交换，得到新的无序区 $(R1,R2....Rn-2)$ 和新的有序区 $(Rn-1,Rn)$ 。不断重复此过程直到无序区的元素个数为1，则整个排序过程完成，存放堆的数组中各元素按照递增顺序排列。

基于大根堆的堆排序最终得到的是一个递增序列，基于小根堆的堆排序最终得到的是一个递减序列。



1.7 字典排序（不重要，刷题看到的）

在数学中，字典或词典顺序（也称为词汇顺序，字典顺序，字母顺序或词典顺序）是基于字母顺序排列的单词按字母顺序排列的方法。对于数字1、2、3..... n 的排列，不同排列的先后关系是从左到右逐个比较对应的数字的先后来决定的。例如对于5个数字的排列12354和12345，排列12345在前，排列12354在后。按照这样的规定，5个数字的所有的排列中最前面的是12345，最后面的是54321。

我们先看一个例子。

示例：1 2 3的全排列如下：

1 2 3 , 1 3 2 , 2 1 3 , 2 3 1 , 3 1 2 , 3 2 1

我们这里是通过字典序法找出来的。

那么什么是字典序法呢？

从上面的全排列也可以看出来了，从左往右依次增大，对这就是字典序法。可是如何用算法来实现字典序法全排列呢？

我们再来看一段文字描述：（用字典序法找124653的下一个排列）

你主要看红色字体部分就行了，这就是步骤。

如果当前排列是124653,找它的下一个排列的方法是，从这个序列中从右至左找第一个左邻小于右邻的数，

如果找不到，则所有排列求解完成，如果找得到则说明排列未完成。

本例中将找到46,记4所在的位置为i,找到后不能直接将46位置互换，而又要从右到左到第一个比4大的数，

本例找到的数是5，其位置计为j，将i与j所在元素交换125643，

然后将i+1至最后一个元素从小到大排序得到125346，这就是124653的下一个排列。

总结得出字典排序算法四步法：

字典排序：

第一步：从右至左找第一个左邻小于右邻的数，记下位置i，值list[a]

第二部：从右边往左找第一个右边大于list[a]的第一个值，记下位置j，值list[b]

第三步：交换list[a]和list[b]的值

第四步：将i以后的元素重新按从小到大的顺序排列

举例：125643的下一个字典序列

第一步：右边值大于左边的 $3 < 4, 4 < 6, 6 > 5$, 则 $i=2$, $list[a]=5$

第二步：从右往左找出第一个右边大于 $list[a]=5$ 的值，找到 $6 > 5, j=3; list[b]=6;$

第三步：交换 $list[a]$ 和 $list[b]$ 的值，序列 $125643 \rightarrow 126543$

第四步：将位置2以后的元素重新排序, $126543 \rightarrow 126345;$

结束：126345即125643的下一个序列

1.8 希尔排序

当需要排序的数据元素比较多时，用希尔排序比用直接插入排序时间更快。

希尔排序其实就是插入排序的一种变种。无论是插入排序还是冒泡排序，如果数组的最大值刚好是在第一位，要将它挪到正确的位置就需要 $n - 1$ 次移动。也就是说，原数组的一个元素如果距离它正确的位置很远的话，则需要与相邻元素交换很多次才能到达正确的位置，这样是相对比较花时间的了。

希尔排序就是为了加快速度简单地改进了插入排序，交换不相邻的元素以对数组的局部进行排序。

希尔排序的思想是采用插入排序的方法，先让数组中任意间隔为 h 的元素有序，刚开始 h 的大小可以是 $h = n / 2$, 接着让 $h = n / 4$, 让 h 一直缩小，当 $h = 1$ 时，也就是此时数组中任意间隔为1的元素有序，此时的数组就是有序的了。

二、设计模式

1.各种原则

1.1开闭原则

对扩展开放，对修改关闭，增加功能是用过增加代码实现的，而不是修改代码

//示例：计算器类要修改就会出错

1.2迪米特法则（最少知识原则）

迪米特法则：最少知识原则，低耦合，高内聚

一个类对于其他类知道的越少越好，就是说一个对象应当对其他对象有尽可能少的了解,只和朋友通信，不和陌生人说话。

1.3合成复用原则

能用组合就别用继承

比如一个汽车类下面有子类大众类、拖拉机类，你如果想要生成一个车手类来开不同的车，就不要用这个车手类去继承大众类或者拖拉机类了（这样只能开一辆车，想要开多辆车就只能再去继承），而是直接在类中定义（组合）一个汽车类指针的成员变量。想开啥车的时候直接让这个指针指向不同的型号的车的类（父类指针指向子类）

1.4 依赖倒转原则

传统的设计方法：倾向于高层模块依赖低层模块。比如说一个银行存款、取款的流程，传统的方式就是在创建一个银行职员类，在类中定义存款、取款的函数来模拟存取款的操作，在高层创建一个通过这个类创建一个银行职员对象要进行存款操作时，要从指定类调用存款成员函数，这样就是高层要依赖底层实现来完成我想要的功能。

依赖倒转原则：将高层依赖低层倒转过来，改为**具体层次依赖于抽象层次**。还是以银行存取款的例子，这里我们把银行职员类定义成一个虚基类，内设一个代表款项操作的函数，然后分别创造两个存款职员类和取款职员类，继承自职员虚基类，并在各自的操作函数中分别执行存取款操作。这样，在高层（业务层）只需要创建一个银行职员虚基类的指针，当想要存款时就把这个指针指向存款子类对象，调用存款函数，想取款时就把这个指针指向取款子类对象，调用取款函数。这样就实现了底层的实现依赖于中间的抽象层。

2.单例

2.1单例的推演

```
//实现单例的步骤
//1.构造函数私有化
//(如果构造函数没有私有化，就没法控制A类型对象的个数，哪个用户
//都可以通过new来调用构造函数创建对象实例)
//2.增加静态私有的当前类的指针变量，让它指向自己的实例
//3.提供静态对外接口，可以让用户获得单例对象
//因为不能使用new创建对象，所以没办法调用public的成员函数
//getInstance()，要想调用它就要使他成为静态函数，这样可以不使用
//具体对象，而是通过类名来调用成员函数，那么私有的A类的指针也要
//是静态的，这个指针指向类new出来的对象，将这个指针作为返回值，
//调用函数就能得到指针变量(也就得到了对象)
```

```
class A{
public:
    static A* getInstance() {
        return a;
    }
private:
    A(){
        a=new A;
    }
    static A* a;
};
```

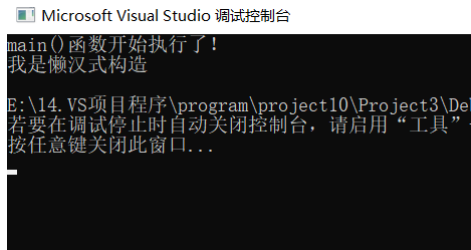
```
//静态变量类内声明，类外初始化
```

```
A* A::a=NULL;  
int main()  
{  
    A::getInstance();  
    return 0;  
}
```

单例模式懒汉代码：

```
#include<stdio.h>  
#include<iostream>  
using namespace std;  
  
//懒汉式  
class Singleton_lazy{  
public:  
    static Singleton_lazy* getInstance() {  
        if (ptr == NULL) {  
            ptr = new Singleton_lazy;  
        }  
        return ptr;  
    }  
private:  
    Singleton_lazy() { cout << "我是懒汉式构造" <<  
endl; }  
    static Singleton_lazy* ptr;  
};  
//类外初始化  
Singleton_lazy* Singleton_lazy::ptr = NULL;  
  
int main()  
{  
    cout << "main()函数开始执行了！" << endl;  
    Singleton_lazy::getInstance();  
    return 0;  
}
```

执行结果：



```
Microsoft Visual Studio 调试控制台
main()函数开始执行了!
我是懒汉式构造

E:\14. VS项目程序\program\project10\Project3\Del
若要在调试停止时自动关闭控制台，请启用“工具”->“
按任意键关闭此窗口...
```

单例模式饿汉代码：

```
#include<stdio.h>
#include<iostream>
using namespace std;

//饿汉式
class Singleton_hungry {
public:

    static Singleton_hungry* getInstance() {
        return ptr;
    }
private:
    Singleton_hungry() { cout << "我是饿汉式构造" <<
endl; }
    static Singleton_hungry* ptr;
};

//类外初始化
Singleton_hungry* Singleton_hungry::ptr = new
Singleton_hungry;

int main()
{
    cout << "main()函数开始执行了! " << endl;
    return 0;
}
```

执行结果：

```
Microsoft Visual Studio 调试控制台
我是饿汉式构造
main() 函数开始执行了!

E:\14.VS项目程序\program\project10\Project3\
若要在调试停止时自动关闭控制台, 请启用“工具
按任意键关闭此窗口...
```

单例和多线程问题，首先给出结论：懒汉式在多线程中不安全，饿汉式在多线程中安全。

2.2 介绍单例及其实现

1. 单例模式定义

保证一个类仅有一个实例，并提供一个访问它的全局访问接口，该实例被所有程序模块共享。

需要保证：

- (1) 该类不能被复制。
- (2) 该类不能被公开的创造。

那么对于C++来说，它的构造函数，拷贝构造函数和赋值函数都不能被公开调用。

2. 单例模式实现方式

单例模式通常有两种模式，分别为**懒汉式单例**和**饿汉式单例**。两种模式实现方式分别如下：

(1) 懒汉式设计模式实现方式（2种）

- a. 静态指针 + 用到时初始化（在getInstance函数里判空创建对象）
- b. 局部静态变量（线程安全）

(2) 饿汉式设计模式（2种）

- a. 直接定义静态对象 *
- b. 静态指针 + 类外初始化时new空间直接创建对象

//懒汉式：只有在调用**getInstance**的时候才会**new**一个对象出来

```
class Singleton_lazy
{
private:
    Singleton_lazy(){}
public:
    static Singleton_lazy* getInstance() {
        if (pSingleton == NULL) {
            pSingleton = new Singleton_lazy;
        }
        return pSingleton;
    }

private:
    static Singleton_lazy* pSingleton;

};
```

//类外初始化

```
Singleton_lazy* Singleton_lazy::pSingleton =
NULL;
```

//饿汉式：在**main**函数开始执行前就已经创造好了

```
class Singleton_hungry
{
private:
    Singleton_hungry() { cout << "我是饿汉构造" <<
endl; }
public:
    static Singleton_hungry* getInstance() {
        return pSingleton;
    }

private:
    static Singleton_hungry* pSingleton;

};
```

```
//类外初始化
Singleton_hungry* Singleton_hungry::pSingleton =
new Singleton_hungry;
```

2.3 懒汉和饿汉区别

1、两者创建单例对象的时机不同，懒汉式是在你程序中真正用到单例对象的时候才会去创建象，而饿汉式是程序在编译阶段就把单例对象创建出来了。解释：懒汉式的单例类中定义的私有静态指向自己单例类实例的那个指针，它在类外被初始化为NULL，而在饿汉式中那个指针被初始化为new这个单例类之后返回回来的指针变量。

2、两者线程安全问题

在多线程情况下，多个线程几乎是同时进行，在懒汉式中多个线程都会进入到Singleton_lazy类中，去调用static Singleton_lazy* getInstance()函数执行ptr = new Singleton_lazy;去创建这个类的实例，所以多个线程都可以去创建类的实例，这样就不是单例模式了，所以懒汉式碰到多线程是不安全的，违背了单例的核心思想。

而饿汉式的对象是在main函数执行前就创建了，也就是说在这些多线程真正启动之前饿汉式就创建了对象，多个线程每次调用static Singleton_hungry* getInstance()函数时，直接就返回了已有对象，所以饿汉式是线程安全的。

2.4 线程安全单例写法

懒汉安全

1.内部静态变量的懒汉单例 (C++11 线程安全)

```
//////////////////// 内部静态变量的懒汉实现
////////////////////
//把函数的声明都放在类内，函数定义放在类外
class Single
{
public:
    // 获取单实例对象
    static Single &GetInstance();

    // 打印实例地址
    void Print();

private:
    // 禁止外部构造
    Single();

    // 禁止外部析构，因为单例模式下我们只有一个对象实例，如果
    // 允许外部用户进行析构，把这个对象删除了，可能会造成程序崩溃
    ~Single();

    // 禁止外部拷贝构造
    Single(const Single &signal);

    // 禁止外部赋值操作
    const Single &operator=(const Single &signal);
    //为什么要禁止赋值操作，如下：
    //Object obj1;
    //Object obj2 = obj1;
    //obj2 = obj1是将obj1拷贝然后赋值给obj2。然后obj1和obj2完
    //全是两个独立的Object，他们的地址不一样的。
    //所以赋值操作也会出现两个实例的情况，所以要把赋值操作放到
    private下
};
```

```

Single &Single::GetInstance()
{
    // 局部静态特性的方式实现单实例
    static Single signal;
    return signal;
}

void Single::Print()
{
    std::cout << "我的实例内存地址是:" << this <<
std::endl;
}

Single::Single()
{
    std::cout << "构造函数" << std::endl;
}

Single::~~Single()
{
    std::cout << "析构函数" << std::endl;
}

////////// 内部静态变量的懒汉实现
//////////

```

C++11规定，当一个线程对静态局部变量进行初始化的时候，其他并发的线程如果也进入到函数想要对这个静态变量初始化的话，其他的线程会被阻塞，直到之前的那个线程完成对静态变量的初始化。所以只有第一个调用方法的线程会完成这个静态局部变量类型的对象的创建，并发线程访问时如果对象没建好会等待，不会重复创建，这样就保证了线程安全。

内部静态变量的懒汉单例的运行结果:

`-std=c++0x` 编译是使用了C++11的特性, 在C++11内部静态变量的方式里是线程安全的, 只创建了一次实例, 内存地址是 `0x6016e8`, 这个方式非常推荐, 实现的代码最少!

```
[root@lincoding singleInstall]#g++ SingleInstance.cpp -o SingleInstance -lpthread -std=c++0x
```

```
main() : 开始 ...
main() : 创建线程:[0]
main() : 创建线程:[1]
Hi, 我是线程 ID:[0]
构造函数
我的实例内存地址是:0x6016e8
Hi, 我是线程 ID:[1]
我的实例内存地址是:0x6016e8
main() : 创建线程:[2]
main() : 创建线程:[3]
main() : 创建线程:[4]
Hi, 我是线程 ID:[3]
我的实例内存地址是:0x6016e8
Hi, 我是线程 ID:[2]
我的实例内存地址是:0x6016e8
main() : 结束!
析构函数
```

局部静态的懒汉式单实例
是线程安全的!

2.双重检测锁

最终实现效果如下

```
if (m_psl == NULL)
{
    // 这里线程开始资源竞争, 只有一个线程能获取lock的资源
    lock();
    if (m_psl == NULL)
    {
        m_psl = new Singleton;
    }
    unlock();
}
```

说明:

lock里面判断一次, 因为可能有多个线程在lock处等待, 一个成功之后, 会将m_psl设置为非空, 这样下个线程就算拿到lock资源, 再进去发现指针非空就离开了

lock外判断一次, 是因为获取锁, 是很浪费时间的, 获取锁之外还有一层判断, 那么在第二次获取单例对象的时候, lock外的if判断发现指针已经非空, 就不会再获取锁了, 直接返回了对应的对象, 这

样双层检测，即保证了对象创建的唯一性，又减少了获取锁浪费的时间和资源

饿汉本身就线程安全

```
//////////////////////////////////// 饿汉实现
////////////////////////////////////
//把函数的声明和变量的定义都放在类内，函数定义放在类外
class Singleton
{
public:
    // 获取单实例
    static Singleton* GetInstance();

    // 释放单实例，进程退出时调用
    static void deleteInstance();

    // 打印实例地址
    void Print();

private:
    // 将其构造和析构成为私有的，禁止外部构造和析构
    Singleton();
    ~Singleton();

    // 将其拷贝构造和赋值构造成为私有函数，禁止外部拷贝和赋值
    Singleton(const Singleton &signal);
    const Singleton &operator=(const Singleton
&signal);

private:
    // 唯一单实例对象指针
    static Singleton *g_pSingleton;
```

```
};

// 代码一运行就初始化创建实例 ，本身就线程安全
Singleton* Singleton::g_pSingleton = new
(std::nothrow) Singleton;

Singleton* Singleton::GetInstance()
{
    return g_pSingleton;
}

void Singleton::deleteInstance()
{
    if (g_pSingleton)
    {
        delete g_pSingleton;
        g_pSingleton = NULL;
    }
}

void Singleton::Print()
{
    std::cout << "我的实例内存地址是:" << this <<
std::endl;
}

Singleton::Singleton()
{
    std::cout << "构造函数" << std::endl;
}

Singleton::~~Singleton()
{
    std::cout << "析构函数" << std::endl;
}

//////////////////////// 饿汉实现
////////////////////////
```


饿汉式单例的运行结果:

从运行结果可知, 饿汉式在程序一开始就构造函数初始化了, 所以本身就线程安全的

☆ 收藏

```
构造函数
main() : 开始 ...
main() : 创建线程:[0]
main() : 创建线程:[1]
Hi, 我是线程 ID:[0]
我的实例内存地址是:main() : 创建线程:[2]
0xf80010
Hi, 我是线程 ID:[1]
我的实例内存地址是:0xf80010
main() : 创建线程:[3]
Hi, 我是线程 ID:[2]
我的实例内存地址是:0xf80010
main() : 创建线程:[4]
Hi, 我是线程 ID:[3]
我的实例内存地址是:0xf80010
析构函数
main() : 结束!
```

代码一运行就初始化创建单实例, 本身就线程安全

3. 工厂

3.1 工厂定义与分类

定义: 定义一个创建对象的接口, 让子类决定实例化哪个类, 而对对象的创建统一交由工厂类去生产, 这样做使得的具体的实现类有更好的封装性, 同时也实现了将用户代码和具体实现类代码之间的解耦, 各做各的事。

分类: 简单工厂模式、工厂方法模式、抽象工厂模式

(1) 简单工厂模式:

它的主要特点是根据传递进来的参数, 在工厂类中做判断, 来确定具体创建哪一个类的对象(产品)。当需要增加新的类对象时, 再修改工厂类。

举例1: 有一家生产水果的厂家, 它只有一个工厂, 可以生产苹果和香蕉, 客户需要什么水果, 只需要告诉生产厂家, 然后生产厂家去生产。(创建一个水果虚基类, 内设一个showName虚函数, 然后分别创建苹果类和香蕉类并继承水果类, 实现各自的showName函数, 然后创建一个工厂类, 在这个工厂类中定义一个生产接口函

数，通过传入参数来创建苹果或者香蕉对象。假如当需要生产苹果时，先生成一个工厂类对象，然后用这个工厂对象调用生产接口去生产不同的水果，并用水果基类指针接住，然后只需要调用这个指针的showName函数既可以知道生产了什么水果)

//简单工厂模式

```
class AbstractFruit {

public:
    virtual void ShowName() = 0;
};

//苹果
class Apple :public AbstractFruit
{
public:
    virtual void ShowName() {
        cout << "我是苹果" << endl;
    }
protected:
private:
};

//香蕉
class Banana :public AbstractFruit
{
public:
    virtual void ShowName() {
        cout << "我是香蕉" << endl;
    }
protected:
private:
};

//鸭梨
```

```

class Pear :public AbstractFruit
{
public:
    virtual void ShowName() {
        cout << "我是鸭梨" << endl;
    }
protected:
private:
};

//水果工厂
class FriutFactorty {
public:
    static AbstractFruit* CreateFruit(string flag) {
        if (flag == "Apple") {
            return new Apple;
        }
        else if (flag == "Banana") {
            return new Banana;
        }
        else if (flag == "Pear") {
            return new Pear;
        }
        else {
            return NULL;
        }
    }
};

void test01() {
    FriutFactorty* factorty = new FriutFactorty;
    AbstractFruit* fruit = factorty-
>CreateFruit("Apple");
    fruit->ShowName();
    delete fruit;
}

```

```

    fruit = factory->CreateFruit("Banana");
    fruit->ShowName();
    delete fruit;

    fruit = factory->CreateFruit("Pear");
    fruit->ShowName();
    delete fruit;

    delete factory;
}

```

举例2：有一家生产处理器核的厂家，它只有一个工厂，能够生产两种型号的处理器核。客户需要什么样的处理器核，一定要显示地告诉生产工厂。下面给出一种实现方案：

```

//程序实例（简单工厂模式）
enum CTYPE {COREA, COREB};
class SingleCore
{
public:
    virtual void Show() = 0;
};
//单核A
class SingleCoreA: public SingleCore
{
public:
    void Show() { cout<<"SingleCore A"<<endl; }
};
//单核B
class SingleCoreB: public SingleCore
{
public:
    void Show() { cout<<"SingleCore B"<<endl; }
};
//唯一的工厂，可以生产两种型号的处理器核，在内部判断
class Factory
{

```

```

public:
    SingleCore* CreateSingleCore(enum CTYPE ctype)
    {
        if(ctype == COREA) //工厂内部判断
            return new SingleCoreA(); //生产核A
        else if(ctype == COREB)
            return new SingleCoreB(); //生产核B
        else
            return NULL;
    }
};

```

优点： 简单工厂模式可以根据需求，动态生成使用者所需类的对象，而使用者不用去知道怎么创建对象，使得各个模块各司其职，降低了系统的耦合性。

缺点： 就是要增加新的核类型时，就需要修改工厂类。这就违反了开放封闭原则：软件实体（类、模块、函数）可以扩展，但是不可修改。

(2)工厂方法模式：

所谓工厂方法模式，是指定义一个用于创建对象的接口，让子类决定实例化哪一个类，也就是把工厂类变成抽象类。（简单工厂中，工厂不符合开闭原则。这里，再将工厂抽象出来，让工厂也符合开闭原则。）

举例1： 现在水果厂商又扩建了一个工厂，原来的工厂专门用来生产香蕉，新工厂生产水果。原来在简单工厂模式中，只有一个工厂类（只有一个工厂），里面通过判断传参来生产不同的水果。而工厂方法则是将这个工厂类也变为虚基类，这个虚基类里有一个生产接口函数（纯虚函数），然后创建苹果工厂类、香蕉工厂类来继承自这个工厂类（有两个工厂），并各自实现生产函数。假如想通过工厂生产苹果了，就先用工厂虚基类创建一个苹果工厂类的对象（父类指针指向子类对象），然后调用这个苹果工厂的生产函数生产苹果，生产出来的苹果对象用水果类的指针接着（父类指针接着）。

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
using namespace std;

//工厂方法模式

class AbstractFruit {

public:
    virtual void ShowName() = 0;
};

//苹果
class Apple :public AbstractFruit
{
public:
    virtual void ShowName() {
        cout << "我是苹果" << endl;
    }
protected:
private:
};

//香蕉
class Banana :public AbstractFruit
{
public:
    virtual void ShowName() {
        cout << "我是香蕉" << endl;
    }
protected:
private:
};

//鸭梨
class Pear :public AbstractFruit
```

```

{
public:
    virtual void ShowName() {
        cout << "我是鸭梨" << endl;
    }
protected:
private:
};

//水果工厂
class FriutFactorty {
public:
    virtual AbstractFruit* CreateFruit() = 0;
};

//苹果工厂
class AppleFactory : public FriutFactorty
{
public:
    virtual AbstractFruit* CreateFruit() {
        return new Apple;
    }
};

//香蕉工厂
class BananaFactory : public FriutFactorty
{
public:
    virtual AbstractFruit* CreateFruit() {
        return new Banana;
    }
};

//鸭梨工厂
class PearFactory : public FriutFactorty
{

```



```
public:
    virtual AbstractFruit* CreateFruit() {
        return new Pear;
    }
};

void test01() {
    FriutFactorty* factorty = new AppleFactory;
    AbstractFruit* fruit = factorty->CreateFruit();
    fruit->ShowName();
    delete fruit;
    delete factorty;

    factorty = new BananaFactory;
    fruit = factorty->CreateFruit();
    fruit->ShowName();
    delete fruit;
    delete factorty;

    factorty = new PearFactory;
    fruit = factorty->CreateFruit();
    fruit->ShowName();
    delete fruit;
    delete factorty;
}

int main()
{
    test01();

    return 0;
}
```

举例2：这家生产处理器核的产家赚了不少钱，于是决定再开设一个工厂专门用来生产B型号的单核，而原来的工厂专门用来生产A型号的单核。这时，客户要做的是找好工厂，比如要A型号的核，就找A工厂要；否则找B工厂要，不再需要告诉工厂具体要什么型号的处理核了。下面给出一个实现方案：

```
//程序实例（工厂方法模式）
class SingleCore
{
public:
    virtual void Show() = 0;
};

//单核A
class SingleCoreA: public SingleCore
{
public:
    void Show() { cout<<"SingleCore A"<<endl; }
};

//单核B
class SingleCoreB: public SingleCore
{
public:
    void Show() { cout<<"SingleCore B"<<endl; }
};

class Factory
{
public:
    virtual SingleCore* CreateSingleCore() = 0;
};

//生产A核的工厂
class FactoryA: public Factory
{
public:
    SingleCoreA* CreateSingleCore() { return new
SingleCoreA; }
};
```

```
//生产B核的工厂
```

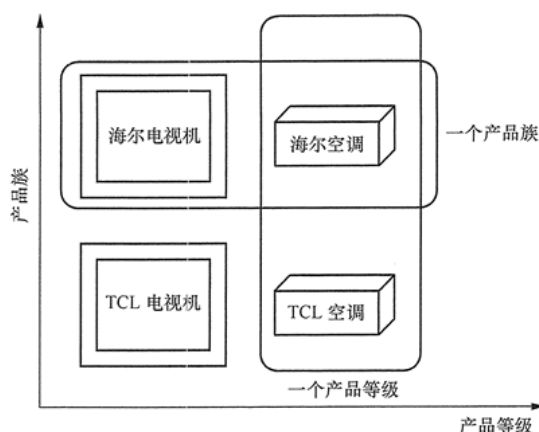
```
class FactoryB: public Factory
{
public:
    SingleCoreB* CreateSingleCore() { return new
    SingleCoreB; }
};
```

优点：扩展性好，符合了开闭原则，新增一种产品时(类对象)，不需要修改原类，只需扩展对应的产品类和工厂子类。

缺点：每增加一种产品，就需要增加一个对象的工厂。如果这家公司发展迅速，推出了很多新的处理器核，那么就要开设相应的新工厂。在C++实现中，就是要定义一个个的工厂类。显然，相比简单工厂模式，工厂方法模式需要定义更多的类(每增加一个新的类，就需要增加一个新的工厂类)，增加维护成本。

(3)抽象工厂模式

工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机，也就是工厂方法模式只考虑生产同等级的产品，抽象工厂模式是考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族。



抽象工厂 (AbstractFactory) 模式的定义：是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

抽象工厂模式是工厂方法模式的升级版，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

使用抽象工厂模式一般要满足以下条件。

- 系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品。
- 系统一次只可能消费其中某一族产品，即同族的产品一起使用。

抽象工厂模式除了具有工厂方法模式的优点外，其他主要优点如下。

- 可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。
- 当需要产品族时，抽象工厂可以保证客户端始终只使用同一个产品的产品组。
- 抽象工厂增强了程序的可扩展性，当增加一个新的产品族时，不需要修改原代码，满足开闭原则。

其缺点是：当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。增加了系统的抽象性和理解难度。

举例1：现在有不同国家（中美倭）的工厂了，然后水果种类也分为中美倭三个国家不同的了（中国苹果、中国香蕉、美国苹果、美国香蕉、倭寇苹果、倭寇香蕉），也就是说现在同一种水果有不同的等级划分了。那么现在要创建三个不同的工厂类，继承自工厂虚基类（中国、美国、倭寇工厂），然后每种水果也要创建三个不同等级的类，这三个类继承自该种水果的虚基类（以苹果为例，建立三个苹果类分别为中国苹果类、美国苹果类、倭寇苹果类，这三个类继承自同一个虚基类苹果类）。这样，中国工厂只负责生产中国的水果，美国工厂只负责生产美国的水果，倭寇工厂只负责生产倭寇水果。

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
```

```
//抽象苹果
class AbstractApple
{
public:
    virtual void showName() = 0;
};

//中国苹果
class ChinaApple : public AbstractApple {
public:
    virtual void showName() {
        cout << "中国苹果" << endl;
    }
};

//美帝苹果
class USAApple : public AbstractApple {
public:
    virtual void showName() {
        cout << "美帝苹果" << endl;
    }
};

//倭寇苹果
class JapanApple : public AbstractApple {
public:
    virtual void showName() {
        cout << "倭寇苹果" << endl;
    }
};

//抽象香蕉
class AbstractBanana
{
public:
    virtual void showName() = 0;
};

//中国香蕉
```

```
class ChinaBanana : public AbstractBanana {
public:
    virtual void showName() {
        cout << "中国香蕉" << endl;
    }
};
//美帝香蕉
class USABanana : public AbstractBanana {
public:
    virtual void showName() {
        cout << "美帝香蕉" << endl;
    }
};
//倭寇香蕉
class JapanBanana : public AbstractBanana {
public:
    virtual void showName() {
        cout << "倭寇香蕉" << endl;
    }
};

//抽象鸭梨
class AbstractPear
{
public:
    virtual void showName() = 0;
};
//中国鸭梨
class ChinaPear : public AbstractPear {
public:
    virtual void showName() {
        cout << "中国鸭梨" << endl;
    }
};
//美帝鸭梨
class USAPear : public AbstractPear {
```

```

public:
    virtual void showName() {
        cout << "美帝鸭梨" << endl;
    }
};

//倭寇鸭梨
class JapanPear : public AbstractPear {
public:
    virtual void showName() {
        cout << "倭寇鸭梨" << endl;
    }
};

//抽象工厂
class AbstractFactory {
public:
    virtual AbstractApple* CreateApple() = 0;
    virtual AbstractBanana* CreateBanana() = 0;
    virtual AbstractPear* CreatePear() = 0;
};

//中国工厂
class ChinaFactory : public AbstractFactory {
public:
    virtual AbstractApple* CreateApple() {
        return new ChinaApple;
    }
    virtual AbstractBanana* CreateBanana() {
        return new ChinaBanana;
    }
    virtual AbstractPear* CreatePear() {
        return new ChinaPear;
    }
};

//美帝工厂
class USAFactory : public AbstractFactory {
public:

```



```

    virtual AbstractApple* CreateApple() {
        return new USAApple;
    }
    virtual AbstractBanana* CreateBanana() {
        return new USABanana;
    }
    virtual AbstractPear* CreatePear() {
        return new USAPear;
    }
};
//倭寇工厂
class JapanFactory : public AbstractFactory {
public:
    virtual AbstractApple* CreateApple() {
        return new JapanApple;
    }
    virtual AbstractBanana* CreateBanana() {
        return new JapanBanana;
    }
    virtual AbstractPear* CreatePear() {
        return new JapanPear;
    }
};

```

```

void test01() {
    //创建一个工厂，依次让中国工厂、美帝工厂、倭寇工厂生产苹果、香蕉、鸭梨
    AbstractFactory* factory = NULL;
    AbstractApple* apple = NULL;
    AbstractBanana* banana = NULL;
    AbstractPear* pear = NULL;

    //中国苹果
    factory = new ChinaFactory;
    apple = factory->CreateApple();
    apple->showName();
}

```

```
delete apple;
//中国香蕉
banana = factory->CreateBanana();
banana->showName();
delete banana;
//中国鸭梨
pear = factory->CreatePear();
pear->showName();
delete pear;
delete factory;

//美帝苹果
factory = new USAFactory;
apple = factory->CreateApple();
apple->showName();
delete apple;
//美帝香蕉
banana = factory->CreateBanana();
banana->showName();
delete banana;
//美帝鸭梨
pear = factory->CreatePear();
pear->showName();
delete pear;
delete factory;

//倭寇苹果
factory = new JapanFactory;
apple = factory->CreateApple();
apple->showName();
delete apple;
//倭寇香蕉
banana = factory->CreateBanana();
banana->showName();
delete banana;
//倭寇鸭梨
pear = factory->CreatePear();
```

```

        pear->showName();
        delete pear;
        delete factory;
    }

    int main(){
        test01();

        return 0;
    }

```

举例2：这家公司的技术不断进步，不仅可以生产单核处理器，也能生产多核处理器。现在简单工厂模式和工厂方法模式都鞭长莫及。抽象工厂模式登场了。它的定义为提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。具体这样应用，这家公司还是开设两个工厂，一个专门用来生产A型号的单核多核处理器，而另一个工厂专门用来生产B型号的单核多核处理器，下面给出实现的代码：

```

//程序实例（抽象工厂模式）
//单核
class SingleCore
{
public:
    virtual void Show() = 0;
};
class SingleCoreA: public SingleCore
{
public:
    void Show() { cout<<"Single Core A"<<endl; }
};
class SingleCoreB :public SingleCore
{
public:
    void Show() { cout<<"Single Core B"<<endl; }
};

```

```

};
//多核
class MultiCore
{
public:
    virtual void Show() = 0;
};
class MultiCoreA : public MultiCore
{
public:
    void Show() { cout<<"Multi Core A"<<endl; }

};
class MultiCoreB : public MultiCore
{
public:
    void Show() { cout<<"Multi Core B"<<endl; }
};
//工厂
class CoreFactory
{
public:
    virtual SingleCore* CreateSingleCore() = 0;
    virtual MultiCore* CreateMultiCore() = 0;
};
//工厂A，专门用来生产A型号的处理器
class FactoryA :public CoreFactory
{
public:
    SingleCore* CreateSingleCore() { return new
SingleCoreA(); }
    MultiCore* CreateMultiCore() { return new
MultiCoreA(); }
};
//工厂B，专门用来生产B型号的处理器
class FactoryB : public CoreFactory
{

```

```
public:
    SingleCore* CreateSingleCore() { return new
SingleCoreB(); }
    MultiCore* CreateMultiCore() { return new
MultiCoreB(); }
};
```

优点： 工厂抽象类创建了多个类型的产品，当有需求时，可以创建相关产品子类和子工厂类来获取。

缺点： 扩展新种类产品时困难。抽象工厂模式需要我们在工厂抽象类中提前确定了可能需要的产品种类，以满足不同型号的多种产品的需求。但是如果我们需要的产品种类并没有在工厂抽象类中提前确定，那我们就需要去修改工厂抽象类了，而一旦修改了工厂抽象类，那么所有的工厂子类也需要修改，这样显然扩展不方便。

1、main()函数调用前会执行一些初始化操作：

包括设置栈指针、

初始化全局变量和静态变量(即data段的内容)、

对未设置初值的全局变量赋初值(数值型short, int, long等为0, bool为FALSE, 指针为NULL等, 即.bss段的内容)、

在main之前调用全局对象的构造函数

给main()函数传递参数, argc, argv

2、

3、

5、

9、

10、

11、函数传参顺序：函数参数是通过栈来传递的，按照被调函数形参顺序相反的顺序入栈，即从右至左，这样参数在执行时就可以按照被调函数原来的顺序从左至右执行

12、(1)如何创建守护进程、(2)僵尸进程是什么，(3)是否所有子进程没被回收都会变成僵尸进程

总结

以下三点总结可以结合前面示例1水果工厂来看

简单工厂是只创建一个工厂，也就是一个工厂类，在其中定义接口函数，根据接口函数传参的不同来判断创建不同的产品（不同的产品继承自同一产品虚基类）。

工厂方法是有多个工厂（不同的工厂继承自同一工厂虚基类），然后每个工厂都只生产一种产品（不同的产品继承自同一产品虚基类）。

抽象工厂也是有多个工厂的，但是对于同一种产品又划分了等级，然后每个工厂只生产本工厂的同族不同等级的产品。（同族不同等级：同族就是同一工厂，比如中国工厂只生产中国产品，不同等级就是说苹果、香蕉这种不同的产品）